Proceedings of Grammar Systems Week 2004, edited by E. Csuhaj-Varjú and Gy. Vaszil, MTA SZTAKI, Budapest, pages 200 - 211, 2004.

# A Possible Connection Between Two Theories: Grammar Systems and Concurrent Programming

María Adela Grando[*]

Research Group on Mathematical Linguistics, Rovira i Virgili University
Pl. Imperial Tárraco 1, 43005 Tarragona, Spain
mariaadela.grando@estudiants.urv.es

Victor Mitrana

Research Group on Mathematical Linguistics, Rovira i Virgili University
Pl. Imperial Tárraco 1, 43005 Tarragona, Spain,
and
Faculty of Mathematics and Computer Science, University of Bucharest
Str. Academiei 14, 70109 Bucharest, Romania
vmi@fll.urv.es

**Abstract**

The aim of this paper is to show how PC grammar systems and concurrent programs might be viewed as related models for distributed and cooperating computation. We argue that it is possible to translate a grammar system into a concurrent program, where one can make use of the Owicki-Gries theory and other tools available in the programming framework. The converse translation is also possible and this turns out to be useful when we are looking for a grammar system able to generate a given language.

In order to show this we use the language: $L_{cd} = \{a^n b^m c^n d^m \mid n, m \geq 1\}$, called *crossed agreement language*, one of the basic non-context free constructions in natural and artificial languages. We prove, using tools from concurrent programming theory, that $L_{cd} \in NPC_3(REG)$ (non-returning PC grammar systems with regular components) solving an open problem introduced in [2].

We also discuss the absence of strategies in the concurrent programming theory to prove that $L_{cd} \notin X_2(REG)$, for $X \in \{PC, CPC, NPC, NCPC\}$, but we prove this in the grammar system framework.

*Keywords:* PC grammar systems, multiprogramming, Owicki-Gries theory.

## 1 Introduction

In the beginning of computation theory, classic computing devices were centralized, that is the computation was accomplished by one central processor. But in modern

[*]This work was possible thanks to the research grant "Programa Nacional para la formación del profesorado universitario", from the Ministry of Education, Culture and Sports of Spain.

computer science distributed computing systems that consist of multiple communicating processors play a major role. The reason is illustrated by the advantages of this kind of system: efficiency, fault tolerance, scalability in the relation between price and performance, etc.

Since 1960, when the concept of *concurrent programming* [1] was introduced, a huge variety of topics related to parallelism and concurrency have been defined and investigated such as operating systems, machine architectures, communication networks, circuit design, protocols for communication and synchronization, distributed algorithms, logics for concurrency, automatic verification and model checking.

The same trend was observed in classic formal language and automata theory as well. In the beginning, grammars and automata were modeling classic computing devices of one agent or processor, hence a language was generated by one grammar or recognized by one automaton. Inspired by different models of distributed systems in Artificial Intelligence, *grammar systems theory* [3] has been developed as a grammatical theory for distributed and parallel computation. More recently, similar approaches have been reported for systems of automata [10].

A grammar system is a set of grammars, working together, according to a specified protocol, in order to generate one language. There are many reasons to consider such a generative mechanism: to model distribution and parallelism, to increase the generative power, to decrease the (descriptional) complexity, etc. The crucial element here is the protocol of cooperation. The theory of grammar systems may be seen as the grammatical theory of cooperation protocols. The central problems are the functioning of systems under specific protocols and the influence of various protocols on various properties of considered systems.

One can distinguish two basic classes of grammar systems: sequential and parallel. In this paper we consider the second class, called *parallel communicating (PC) grammar system* [14]. In the next section, we recall the basic definitions related to this model as well as the basic concepts of concurrent programming and Owicki-Gries theory which is known as the first complete programming logic used for formal development of concurrent programs.

*Owicki-Gries theory* [11] and other strategies of programming were developed to help programmers in the analysis and design of multiprograms. In this paper we argue that grammar system theory can benefit from the tools already developed in the programming framework. For example, given a grammar system one can prove that it generates a specific language by a direct reasoning or one can translate the grammar system into a multiprogram and prove the same statement by some strategies of programming developed in the well known Owicki-Gries theory. Furthermore, we propose another approach to solve problems of the following type: *Given a language find a grammar system that generates the given language.* The strategy widely used so far is as follows: first one proposes a grammar system and then proves by means of language theory that the proposed grammar system generates indeed the given language. We give an example of how Owicki-Gries logic of programming could guide us in obtaining, *simultaneously*, the grammar systems that generates the given language and a proof that it really generates it. This new approach might be of a great benefit for the grammar systems theory. We apply this strategy for a well-known non-context-free language, namely $L_{cd} = \{a^n b^m c^n d^m \mid n, m \geq 1\}$.

In [2] it was proved that $L_{cd} \in CPC_4(CF)$ (see the next section for notations), we improved this result showing that $L_{cd} \in NPC_5(REG)$, providing thus a solution to an open problem mentioned in that work. During the workshop, G. Păun proposed us a more economical solution with respect to the number of components of the grammar system, namely $L_{cd} \in NPC_3(REG)$, based on a similar strategy. We give here, in Section 3, the solution proposed by Păun. As we shall see in the same section, this is actually the most economical solution. The strategy consists in translating the problem of finding a non-returning, non-centralized PC grammar system $\Gamma$ with regular components that generates $L_{cd}$, into the problem of finding a multiprogram $\mathcal{P}$, with three programs $Prog_i$, $i = 1, 2, 3$, running concurrently, which is correct with respect to the specification:

$$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge (w_3 = S_3)\}\mathcal{P}\{w_1 \in L_{cd}\}.$$

Then this multiprogram is translated back into a PC grammar system $\Gamma$ with three regular components, the whole behavior of $\Gamma$ being similar to that of $\mathcal{P}$. Actually, this means that the language generated by $\Gamma$ is included in $L_{cd}$ but the detailed reasoning presented in the third section allows us to conclude the equality.

In this paper we also show that though PC grammar systems theory can benefit from concurrent programming theory to get positive results, the later theory cannot provide any strategy to deal with negative results of the type: *A given language cannot be generated by any grammar system of a specified type.* This kind of problems has to be attacked in the grammar system framework, with the tools available there. We exemplify this with a proof that $L_{cd} \notin NPC_2(REG)$.

## 2 Preliminaries

An *alphabet* is a finite and nonempty set of symbols. Any sequence of symbols from an alphabet $V$ is called *word* over $V$. The set of all words over $V$ is denoted by $V^*$ and the empty word is denoted by $\lambda$. Further, $V^+ = V \setminus \{\lambda\}$.

For all unexplained notions the reader is referred to [15].

### 2.1 PC Grammar Systems

A *PC grammar system* of degree $n, n \geq 1$, is an $(n+3)$-tuple

$$\Gamma = (N, K, T, (P_1, S_1), (P_2, S_2), (P_3, S_3), ...., (P_n, S_n)),$$

where:

- $N$ is a nonterminal alphabet,
- $T$ is a terminal alphabet,
- $K = \{Q_1, Q_2, ....., Q_n\}$ (the sets $N, T, K$ are mutually disjoint),
- $P_i$ is a finite set of rewriting rules over $N \cup K \cup T$, and $S_i \in N$, for all $1 \leq i \leq n$.

Let $V_\Gamma = N \cup K \cup T$. The sets $P_i$, $1 \leq i \leq n$, are called the components of the system, and the elements $Q_1, Q_2, ....., Q_n$ of $K$ are called query symbols, the index $i$ of $Q_i$ points to the component $P_i$ of $\Gamma$. A component is said to be regular if all its rules are right-linear ones. An $n$-tuple $(x_1, x_2, ....., x_n)$ with $x_i \in V_\Gamma^*$ for all $i$, $1 \leq i \leq n$, is called a *configuration* of $\Gamma$. A configuration $(x_1, x_2, ....., x_n)$ directly yields another configuration $(y_1, y_2, ......., y_n)$ if either:

- No query symbols appears in $x_1, x_2, ....., x_n$ and then we have a componentwise derivation, $x_i \Longrightarrow y_i$ in each component $P_i$, $1 \leq i \leq n$ (one rule is used in each component $P_i$), except for the case when $x_i$ is terminal, $x_i \in T^*$; then $x_i = y_i$, or
- Query symbols occur in some $x_i$. Then a *communication step* is performed: every $x_i$ (containing query symbols) is modified by substituting $x_j$ for each occurrence of a query symbol $Q_j$, providing $x_j$ does not contain query symbols. After all words $x_i$ have been modified, the component $P_j$ continues its work on the current string (in the *non-returning* case) or resumes working from its axiom (in the *returning* case. The communication has priority over the effective rewriting: no rewriting is possible as long as at least one query symbol is present. If some query symbols are not satisfied at a given moment, then they have to be satisfied as soon as other query symbols have been satisfied.

If only the first component is entitled to introduce query symbols, then the system is called *centralized.*

The *language generated by a PC grammar system* $\Gamma$ as above is

$$L(\Gamma) = \{x \in T^* \mid (S_1, S_2, ....S_n) \Longrightarrow (x, \alpha_2, ....., \alpha_n), \alpha_i \in V_\Gamma^*, 2 \leq i \leq n\}.$$

Hence, one starts from the $n$-tuple of axioms, $(S_1, S_2, ....S_n)$, and proceeds by repeated rewriting and communication steps, until the component $P_1$ produces a terminal string. The component $P_1$ is called the *master* of the system.

The class of languages generated by non-centralized, centralized, non-returning non-centralized, non-returning centralized PC grammar systems with $k$ regular components is denoted by $PC_k(REG)$, $CPC_k(REG)$, $NPC_k(REG)$, and $NCPC_k(REG)$, respectively.

## 2.2 Programming

### 2.2.1 Sequential Programming

A sequential program consists in:

– A number of declarations,

– A sequence of instructions or actions.

The actions take place one after another. That is, an action does not begin until the preceding one has ended. Because a sequential program has a sequence of actions we consider a program as a transformer of states or predicates (see, e.g., [4] and [7]), where a state $\{P\}$ describes the relationships between the variables of the systems and their values by the predicate $P$ . Each action $\mathcal{S}$ transforms the current state of the system, called precondition of $\mathcal{S}$, to the state $\{Q\}$ which is called postcondition.

A *Hoare triple* is a sequence $\{P\}\,\mathcal{S}\,\{Q\}$ , where:

– $\mathcal{S}$ is an action or instruction,
– $\{P\}$ is a state representing the precondition of $\mathcal{S}$,
– $\{Q\}$ is a state representing the postcondition of $\mathcal{S}$.

Its operational interpretation is as follows: $\{P\}\,\mathcal{S}\,\{Q\}$ is a correct Hoare triple if and only if it is true that each terminating execution of $\mathcal{S}$ that starts from a state satisfying $P$ is guaranteed to end up in a state satisfying $Q$. More precisely, if $\{P\}\,\mathcal{S}\,\{Q\}$ holds and $\mathcal{S}$ starts in a state satisfying $P$, we can be sure that $\mathcal{S}$ either terminates in a state satisfying $Q$ or does not terminate at all. Consequently, a program ought to be annotated in such a way that each action carries a precondition. In other words, from a logical perspective a sequential program may be viewed as a sequence of Hoare triples.

We can now formulate the concept of *local correctness* of a predicate $Q$ in a program. We distinguish two cases:

– If $Q$ is the initial predicate of the program, it is locally correct whenever it is implied by the precondition of the program as a whole. Also we may say that $Q$ satisfies the hypothesis of the problem which is to be solved.

– If $Q$ is preceded by $\{P\}\,\mathcal{S}$, i.e. by atomic action $\mathcal{S}$ with precondition $P$, it is locally correct whenever $\{P\}\,\mathcal{S}\,\{Q\}$ is a correct Hoare-triple.

A sequential program is *partially correct* if all its predicates are locally correct and the last predicate satisfies the requirements of the problem solved, provided that it halts. A sequential program is *totally correct* if it is partially correct and always halts.

### 2.2.2 Concurrent Programming

Concurrent execution or multiprogramming means that various sequential programs run simultaneously. Actions change the state of the multiprogram, so the critical question now is what happens if two overlapping actions change the same state of the multiprogram in a conflicting manner.

Now we are ready to formulate what we call the Core of the **Owicki-Gries theory** (see [10]). We consider a multiprogram annotated in such a way that the annotation provides a precondition for the multiprogram as a whole and a precondition for each action in each individual program. Then, by Owicki and Gries, this annotation is correct whenever each individual predicate is correct, i.e.:

– locally correct as described above and

– globally correct. Predicate $Q$ in a multiprogram $\mathcal{M}$ is *globally correct* whenever for each $\{P\}\,\mathcal{S}$, i.e. for each action $\mathcal{S}$ with precondition $P$, taken from a program of $\mathcal{M}$, $\{P \wedge Q\}\,\mathcal{S}\,\{Q\}$ is a correct Hoare-triple.

To understand how powerful is the concurrent programming, and also how hard is to prove global correctness we give this simple example:

**Example 1** *Consider this program:*

$$\begin{array}{rrcl} P_1: & x := & y+1; & \quad a \\ & x := & y^2; & \quad b \\ & x := & x-y & \quad c \end{array}$$

If we start in an initial state $\{x = 7 \wedge y = 3\}$, it will deliver $\{x = 6 \wedge y = 3\}$ as a final state.

**Example 2** *Also consider this simple program:*

$$\begin{array}{llll} P_2: & y := & x+1; & u \\ & y := & x^2; & v \\ & y := & y-x & w \end{array}$$

When started in the same initial state $\{x = 7 \wedge y = 3\}$, it yields $\{x = 7 \wedge y = 42\}$.

Now if we run these programs concurrently we will get 20 possible values for $x$ and $y$. For instance, one possibility is to run the two programs as follows: $a, u, b, v, w, c$ (the letters represent the program lines) starting from the same state and get the output $\{x = -224, y = 240\}$. While each of the individual programs is of an extreme simplicity, their composition leads to a rather complicated output. For more examples we refer to [5].

Here we can see the analogy: the components of a grammar systems are similar to the simple programs in a multiprogram. We introduce a proof that shows how to take advantage of this analogy.

## 3 Main Result

**Theorem 1** $L_{cd} \in NPC_3(REG)$

*Proof.* We want to find a non-returning, non-centralized grammar system $\Gamma$ with regular components that generates $L_{cd}$. This problem is transformed into the equivalent problem of finding a multiprogram $\mathcal{P}$ that behaves like $\Gamma$ and is correct with respect to the specification:

$$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge \ldots\ldots \wedge (w_n = S_n) \mid n \geq 1\}\mathcal{P}\{w_1 \in L_{cd}\}.$$

The problem remained the same, but we changed the tools to solve it: instead of *induction* and *analysis by cases* available in the framework of grammar systems we used *Logic, Owicki-Gries theory* and *programming strategies* from the programming framework.

The strategy used for this proof is one frequently used for the development of programs, called *refinement of the problem* that consists in:

1. First, start with an outline of the solution, which identifies the basic principle by which the input can be transformed into the output. Define pre and post conditions for each of the subproblems that are identified as part of the solution for the hole problem.

For our problem we proposed this idea:

$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge \ldots\ldots \wedge (w_n = S_n)\}$

**Subproblem 1***: (Rewrite)$^p$, with $p \geq 1$*

$\{(w_1 = S_1) \wedge \ldots\ldots\ldots \wedge (w_i = a^p S_i) \wedge \ldots\ldots \wedge (w_j = c^p S_j) \wedge \ldots\ldots \wedge (w_n = S_n) \wedge (p \geq 1)\}$

**Subproblem 2***: (Rewrite; Communication)$^+$*

Find a way to stop the productions of *a's* and *c's*, through synchronization by communication.

$\{(w_1 = a^r N_1) \wedge ....... \wedge (w_k = c^r N_2) \wedge ...... \wedge (w_n = S_n) \wedge (r \geq 1) \wedge (N_1, N_2 \in N)\}$

**Subproblem 3***: (Rewrite)$^m$, with $m \geq 1$*

$$\left\{ \begin{array}{c} (w_1 = a^r b^m Q_k) \wedge ....... \wedge (w_k = c^r d^{m-1} N_3) \wedge ...... \wedge (w_n = S_n) \wedge \\ (r, m \geq 1) \wedge (Q_k \in K) \wedge (N_3 \in N) \end{array} \right\}$$

**Subproblem 4***: Communication*

$\{(w_1 = a^r b^m c^r d^{m-1} N_3) \wedge (r, m \geq 1) \wedge (N_3 \in N)\}$

**Subproblem 5***: Rewrite*

$\{(w_1 = a^r b^m c^r d^m) \wedge (r, m \geq 1)\}$

or equivalently

$\{w_1 \in \{a^r b^m c^r d^m \wedge r \geq 1 \mid m \geq 1\}\}$

2. Now we make precise the outline indicated, refine the subproblems trying to find simultaneously the instructions that solve the subproblems and the proof of its local correctness. We also discuss the difficulties we can have when proving global correctness.

In our refinement of subproblems 1, 2, 3, 4 and 5 we proposed three programs $Prog_1$, $Prog_2$ and $Prog_3$. These programs forming the multiprogram $P$, run simultaneously behaving like a non-returning, non-centralized grammar system with regular productions and behave locally correctly with respect to the subproblems that we have identified in the previous step.

In the case of Subproblem 1 we propose this refinement:

$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge (w_3 = S_3)\}$

**Subproblem 1***: Rewrite$^n$, with $n \geq 1$*

$Prog_1$ rewrites $n-1$ times $S_1$ to $aS_1$ and then rewrites $S_1$ to $aA$, $Prog_2$ rewrites $n-1$ times $S_2$ to $cS_2$ and then rewrites $S_2$ to $cB$ and $Prog_3$ rewrites $n-1$ times $S_3$ to $S_3$, until it decides to finish the production of *a's* and *c's*, rewriting $S_3$ to $Q_2$.

To be sure that $w_2 = c^n B$ when $Prog_3$ introduces $Q_2$, $Prog_3$ should not be able to rewrite $S_2$, and after $Prog_2$ introduces $B$ it should rewrite it for another nonterminal and not introduce $B$ anymore.

The reason why $w_1 = a^n A$ and $w_1 \neq a^n S_1$ is that this is the only possibility that do not lead to deadlock, as the states of the next subproblem show.

$\{(w_1 = a^n A) \wedge (w_2 = c^n B) \wedge (w_3 = Q_2) \wedge (n \geq 1)\}$

For Subproblem 2 we propose this sequence of rewritings and communications as a refinement:

$\{(w_1 = a^n A) \wedge (w_2 = c^n B) \wedge (w_3 = Q_2) \wedge (n \geq 1)\}$

**Subproblem 2**

*Communication*
$\{w_1 = a^n A \wedge w_2 = c^n B \wedge w_3 = c^n B \wedge n \geq 1\}$
*Rewrite*
*Prog*1 rewrites $A$ to $A$', *Prog*2 rewrites $B$ to $Q_1$ and
*Prog*3 rewrites $B$ to $D$
We do not allow other possibility than
$w_1 = a^n A' \wedge w_2 = c^n Q_1 \wedge w_3 = c^n D$.
To be sure that $w_1 = a^n A'$ after the rewriting step,
we need $Prog_2$ to be only defined for $A$', and after $Prog_1$
introduces $A$' it should rewrite it to another
nonterminal and not introduce $A$' anymore.
$\{w_1 = a^n A' \wedge w_2 = c^n Q_1 \wedge w_3 = c^n D \wedge n \geq 1\}$
*Communication*

$\{(w_1 = a^n A') \wedge (w_2 = c^n a^n A') \wedge (w_3 = c^n D) \wedge (n \geq 1)\}$

In the case of Subproblem 3 this is a possible refinement:
$\{(w_1 = a^n A') \wedge (w_2 = c^n a^n A') \wedge (w_3 = c^n D) \wedge (n \geq 1)\}$
**Subproblem 3***: Rewrite$^{m+1}$, with* $m \geq 1$
$Prog_1$ rewrites $A$' to $A$" and rewrites $m-1$ times $A$" to $bA$", and then rewrites $A$" to $bQ_3$, $Prog_2$ always rewrites $A$' to $A$' and *Prog*3 rewrites $D$ to $D$', then $D$' to $D$" and rewrites $m-1$ times $D$" to $dD$"
$\{(w_1 = a^n b^m Q_3) \wedge (w_2 = c^n a^n A') \wedge (w_3 = c^n d^{m-1} D") \wedge (n, m \geq 1)\}$

Refinement for Subproblem 4 and Subproblem 5 is very simple:
$\{(w_1 = a^n b^m Q_3) \wedge (w_2 = c^n a^n A') \wedge (w_3 = c^n d^{m-1} D") \wedge (n, m \geq 1)\}$
**Subproblem 4:** *Communication*
$\{(w_1 = a^n b^m c^n d^{m-1} D") \wedge (w_2 = c^n a^n A') \wedge (w_3 = c^n d^{m-1} D") \wedge (n, m \geq 1)\}$
**Subproblem 5:** *Rewrite*
$Prog_1$ rewrites $D$" to $d$
$\{(w_1 \in \{a^n b^m c^n d^m) \wedge (n, m \geq 1)\}\}$
Equivalently we proposed a non-returning, non-centralized grammar system $\Gamma$ with three regular components, defined in this way:

$$\Gamma = (N, K, \{a, b, c, d\}, (P_1, S_1), (P_2, S_2), (P_3, S_3))$$

where:
$N = \{S_1, S_2, S_3, A, A', A", B, D, D', D"\}$
$K = \{Q_1, Q_2, Q_3\}$
$P_1 = \{S_1 \longrightarrow aS_1, S_1 \longrightarrow aA, A \longrightarrow A', A' \longrightarrow A", A" \longrightarrow bA", A" \longrightarrow bQ_3,$
$D" \longrightarrow d\}$
$P_2 = \{S_2 \longrightarrow cS_2, S_2 \longrightarrow cB, B \longrightarrow Q_1, A' \longrightarrow A'\}$
$P_3 = \{S_3 \longrightarrow S_3, S_3 \longrightarrow Q_2, B \longrightarrow D, D \longrightarrow D', D' \longrightarrow D", D" \longrightarrow dD"\}$.

3. The last and hardest step is to prove global correctness. In our case it means that we have to show using Owicki-Gries theory that the multiprogram $\mathcal{P}$ we constructed satisfies this specification:

$$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge (w_3 = S_3)\}\mathcal{P}\{w_1 \in L_{cd}\}.$$

Furthermore, $\mathcal{P}$ outputs the word $a^n b^m c^n d^n$ for any input formed by the pair of positive integers $n, m$. This is equivalent to prove that $L(\Gamma) = L_{cd}$.

According to how we defined $Prog_1$, $Prog_2$ and $Prog_3$, behaving like $G_1$, $G_2$ and $G_3$, respectively, we propose the following state:

$$\left\{\begin{array}{c} \left[\begin{array}{c} (w_1 = a^n S_1 \wedge n \geq 0) \vee (w_1 = a^n A \wedge n \geq 1) \vee (w_1 = a^n A' \wedge n \geq 1) \vee \\ \vee (w_1 = a^v b^n A'' \wedge v \geq 1 \wedge n \geq 0) \vee (w_1 = a^v b^n Q_3 \wedge v \geq 1 \wedge n \geq 1) \vee \\ \vee (w_1 = a^v b^n c^g d^h D'' \wedge v, n, g \geq 1 \wedge h \geq 0) \vee (w_1 = a^e b^f c^g d^h \wedge e, f, g, h \geq 1) \end{array}\right] \wedge \\ \wedge \left[\begin{array}{c} (w_2 = c^q S_2 \wedge q \geq 0) \vee (w_2 = c^q B \wedge q \geq 1) \\ (w_2 = c^q Q_1 \wedge q \geq 1) \vee (w_2 = c^q a^r A' \wedge q, r \geq 1) \end{array}\right] \wedge \\ \wedge \left[\begin{array}{c} (w_3 = S_3) \vee (w_3 = Q_2) \vee (w_3 = c^n B \wedge n \geq 1) \vee (w_3 = c^n D \wedge n \geq 1) \vee \\ \vee (w_3 = c^n D' \wedge n \geq 1) \vee (w_3 = c^n d^m D'' \wedge n \geq 1 \wedge m \geq 0) \end{array}\right] \end{array}\right\}$$

But by Owicki-Gries theory of global correctness one can prove that after $n$ rewritings, with $n \geq 1$, the only possible combination of values for the sentential forms $w_1$, $w_2$ and $w_3$ that do not lead to a deadlock, is the one expressed by the state:

$$\{(w_1 = a^n A) \wedge (w_2 = c^n B) \wedge (w_3 = Q_2) \wedge (n \geq 1)\}$$

And from this state it can be proved that the only valid continuation is the sequence of rewritings and communications described in step 2 of the refinement process, that reaches to the state containing $\{w_1 \in \{a^n b^m c^n d^m \mid n, m \geq\}\}$ $\square$

We have proved in the framework of programming that it is possible to generate $L_{cd}$ with a grammar system with three components with regular productions, working in non-returning, non-centralized way. The strategy we have presented differs from the traditional approach not in complexity, because the number of cases considered in the proofs are the same, but in the way of reasoning about the problem. We state that Owicki-Gries methodology provides more possibilities for reasoning about problems, in comparison with the strategies used so far in grammar system framework because:

- it allows to reason in a forward or data-driven way, as analysis by cases techniques, but also in a backward or goal-directed way.

The notion of backward reasoning comes from psychology, as it is pointed in [9] where this description of problem solving occurs: *We may have a choice between starting with where we wish to end, or starting with where we are at the moment. In the first instance we start by analyzing the goal. We ask, "Suppose we did achieve the goal, how would things be different- what subproblems would we have solved, etc.?". This in turn would determine the sequence of problems, and we would work back to the beginning. In the second instance we start by analyzing the present situation, see the implications of the given conditions and lay-out, and attack the various subproblems in a "forward direction".*

- the division of problems in subproblems is possible because of the theorem: for any $Q$ $\{P\}\mathcal{S}_0; \mathcal{S}_1\{R\} \Longleftarrow \{P\}\mathcal{S}_0\{Q\} \wedge \{Q\}\mathcal{S}_1\{R\}$, where $P$,$R$ are predicates and $\mathcal{S}_0, \mathcal{S}_1$ are instructions. Also goals and subgoals are discussed in the psychology text mentioned above ([9]): *The person perceives in his surrounding goals capable of removing his needs and fulfilling his desires... And there is the important phenomenon of emergence of subgoals. The pathways to goals are often perceived as organized* *into a number of subparts, each of which constitutes and intermediate subgoal to be attained on the way to the ultimate goal.*

These characteristics make Owicki-Gries strategies more related with human way of reasoning.

Now we want to prove a negative result about grammar systems: There is no grammar system of any type with two regular components that can generate $L_{cd}$. In other words, the solution we proposed is the most economical one with respect to the number of components. If we translate this problem into the programming framework, we have to prove that it is not possible to find a multiprogram $\mathcal{P}$ with two programs $\mathcal{P}_1$ and $\mathcal{P}_2$ running concurrently, modifying $w_1$ and $w_2$ in a right-linear way, that is correct with respect to this specification: $\{(w_1 = S_1) \wedge (w_2 = S_2)\}\mathcal{P}\{w_1 \in \{a^n b^m c^n d^m \mid n, m \geq\}\}$. But unfortunately we have no strategy or result in the concurrent programming theory that could help us to reasoning about this problem. The only strategies available in this framework are: *verification* that allows to prove the correctness of a multiprogram with respect to a specification, and the *constructive approach* that was exemplified in the previous proof. But these strategies are useful to get positive results therefore we have to remain in the grammar system framework to deal with this kind of problems. We solve it with the tools available there, namely analysis by cases.

**Theorem 2** $L_{cd} \notin X_2(REG)$, *for* $X \in \{PC, CPC, NPC, NCPC\}$.

*Proof.* Since $PC_2(REG)$ (hence $CPC_2(REG)$, too), contains context-free languages only ([3]), it suffices to prove that $L_{cd}$ cannot lie in $NPC_2(REG)$. Assume that $L_{cd} = L(\Gamma)$ for some non-returning non-centralized grammar system with two regular components $\Gamma$. Take $w = a^n b^m c^n d^m$ with arbitrarily large $n, m$. There exist two nonterminals $A_1, A_2$ such that in the process of generating $w$ the following hold:

$$(S_1, S_2) \Longrightarrow^* (x_1 A_1, x_2 A_2), x_1, x_2 \in \{a, b, c, d\}^*,$$
$$(A_1, A_2) \Longrightarrow^{k_1} (u A_1, v A_2), u, v \in \{a, b, c, d\}^*, uv \neq \lambda$$

for some $k_1 \geq 1$. Here $\Longrightarrow^p$ denotes a derivation of length $p$ where the communication steps are also counted. Let $(A_1, A_2)$ be the first pair of such nonterminals met in the process with this property. By the choice of $w$ we infer that $u \in a^+$ and $v \in c^+$. First we note that both $u$ and $v$, if non-empty, are formed by one letter only. Second, if one is empty, then the other can be "pumped" arbitrarily many times which leads to a parasitic word. Third, by the same argument, all the other choices, except $u \in a^+$ and $v \in c^+$, lead to parasitic words. Since the subword of $w$ formed by $b$ is arbitrary long there is a pair of nonterminals $(B_1, B_2)$ such that the derivation continues as follows:

$$(S_1, S_2) \Longrightarrow^* (x_1 A_1, x_2 A_2) \Longrightarrow^{rk_1} (x_1 u^r A_1, x_2 v^r A_2) \Longrightarrow^k$$
$$(x_1 u^r y_1 B_1, x_2 v^r y_2 B_2) \Longrightarrow^{pk_2} (x_1 u^r y_1 s^p B_1, x_2 v^r y_2 t^p B_2) \Longrightarrow^* (w, \alpha)$$

for some terminal words $y_1, y_2, s, t,\ s \in b^+$, positive integers $r, p, k, k_2$, and word $\alpha$. Moreover, no communication step appears in the subderivation

$$(x_1 u^r A_1, x_2 v^r A_2) \Longrightarrow^* (x_1 u^r y_1 B_1, x_2 v^r y_2 B_2) \Longrightarrow^{pk_2} (x_1 u^r y_1 s^p B_1, x_2 v^r y_2 t^p B_2)$$

otherwise either an insufficient number of $b$'s is generated between the two subwords formed by $a$ and $c$ or the generated word contains the subword $cb$ which is contradictory.

Therefore, the following derivation is also possible in $\Gamma$:

$$(S_1, S_2) \Longrightarrow^* (x_1A_1, x_2A_2) \Longrightarrow^{rk_1} (x_1u^rA_1, x_2v^rA_2) \Longrightarrow^{k_1k_2+k}$$
$$(x_1u^{r+k_2}y_1B_1, x_2v^ry_2t^{k_1}B_2) \Longrightarrow^{pk_2} (x_1u^{r+k_2}y_1s^pB_1, x_2v^ry_2t^{p+k_1}B_2) \Longrightarrow^* (w', \alpha'),$$

for terminal word $w'$ and word $\alpha'$. However, $w'$ cannot be in $L_{cd}$, which concludes the proof. □

We encourage the reader to translate this proof into the programming framework, where the reasoning is the same, but more explanations are needed.

## 4 Conclusions

The traditional approach to the problem of finding a grammar system generating a given language is: first propose a grammar system and then find a proof that it generates the language.

In this paper we present a new approach, taken from programming framework, that consists in finding simultaneously the grammar system that generates the given language and a proof that the grammar system found generates it. We think that it would be interesting to study this approach deeper trying to apply it to other well-known languages, or trying to find other programming strategies, apart from the strategy of refinement of problems shown here, that could be useful in solving some problems related to grammar system theory.

Until now main efforts in grammar system theory have been dedicated to find grammar systems with the smallest number of components or different types of restrictions applied to productions, to show how distribution and communication can make simple components be very powerful when they work together. So except for some studies related with computational complexity measure of PC grammar systems that considers the number of communication between grammars (see for example [8] and [13]), the most investigated complexity measure has been the number of grammars that a PC grammar system consist of, which is clearly a descriptional complexity measure. So a very important matter has been forgotten: the efficient use of time. The opposite has happened in the programming area (see [6] and [12]), where the research has been focused in looking for techniques to parallelize algorithms and to help programmers to design more efficient concurrent algorithms. We propose to follow some of the methodical approaches developed in the programming framework to construct more efficient grammar systems.

Also it would be very interesting not only to think how PC grammar system theory can benefit from concurrent programming, but how programming theory can benefit from grammar system theory. The lack of strategies in the programming framework to prove negative results of the type: $L \neq L(\Gamma)$ for a language $L$ and any grammar system $\Gamma$, make us to think that such problems might be solved by translating them into the grammar system framework where they can be solved using the tools available there.

## References

[1] A. Burns, G. Davies, *Concurrent Programming*, Addison-Wesley, Wokingham, England (1993).

[2] A. Chitu, PC grammar systems versus some non-context free constructions from natural and artificial languages, *New Trends in Formal Languages* (Gh. Păun, A. Salomaa, eds.), LNCS 1218, Springer-Verlag, Berlin (1997) 278-287.

[3] E. Csuhaj-Varjú, J. Dassow, J. Kelemen, Gh. Păun, *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*, Gordon and Breach, London, (1994).

[4] Dijkstra, W. Edsger, *A Discipline of Programming*, Prentice-Hall Series in Automatic Computation (1976).

[5] W. H. J. Feijen, A. J. M. van Gasteren (Eds.), *On A Method of Multi-Programming*, Springer-Verlag, Berlin (1999).

[6] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley (1995).

[7] D. Gries, *The Science of Programming*, Springer-Verlag, Berlin (1981).

[8] J. Hromkovič, J. Kari, L. Kari, Some hierarchies for the communication complexity measures of cooperating grammar systems, *Theoretical Computer Science* 127, 1 (1994) 123-147.

[9] D. Krech, R. S. Cruthfield, *Elements of Psycology*, Knopf, New York (1958) 383.

[10] C. Martin-Vide, V. Mitrana, Parallel communicating automata systems- a survey, *Korean J. Comput. Appl. Math.* 7, 2 (2000) 237-257.

[11] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs 1, *Acta Informatica* 6 (1976) 319-340.

[12] D. Parnas, P. Clements, A rational design process: How and why to fake it. *IEEE Trans. Software Eng.* SE-12(2) (1986) 251–257.

[13] D. Pardubská, On the power of communication structure for distributive generation of languages, *Developments in Language Theory, At the Crossroads of Mathematics, Computer Science and Biology* (G. Rozenberg, A. Salomaa, eds.), World Scientific, Hugapore (1993) 419-429.

[14] Gh. Păun, L. Sântean, Parallel communicating grammar systems: the regular case, *Ann. Univ. Buc., Ser. Matem.-Inform*, 38 (1989) 55-63.

[15] G. Rozenberg, A. Salomaa (Eds.), *Handbook of Formal Languages*, Springer-Verlag, Berlin (1997).